Dataclass

В предыдущем примере мы работали с Mongo с помощью словарей.

В главе про aiohttp мы говорили, что лучше описывать данные в коде, чтобы понимать, какая у них вообще структура.

В Mongo это еще более актуально. В каждый момент времени нужно понимать, с каким объектом ты работаешь, какие поля у него есть и что вообще он умеет делать. Поэтому очень важно описывать в коде структуры документов, которые хранятся в коллекциях Mongo.

Пример: опишем структуру коллекции user.

connectors/mongo/schema.py

```
import typing

import bson
from dataclasses import field
from marshmallow import Schema, ValidationError, fields, missing
from marshmallow_dataclass import dataclass


class ObjectIdField(fields.Field):
    def _deserialize(
            self,
            value: typing.Any,
            attr: typing.Optional[str],
            data: typing.Optional[typing.Mapping[str, typing.Any]],
            **kwargs,
    ):
        try:
            return bson.ObjectId(value)
        except Exception:
            raise ValidationError("invalid ObjectId `%s`" % value)

    def _serialize(self, value: typing.Any, attr: str, obj: typing.Any, **kwargs):
        if value is None:
            return missing
        return str(value)


Schema.TYPE_MAPPING[bson.ObjectId] = ObjectIdField


@dataclass
class User:
    first_name: str
    last_name: str
    is_tutor: bool
    _id: typing.Optional[bson.ObjectId] = field(default=None)

    Schema: typing.ClassVar[typing.Type[Schema]] = Schema
```

Разберемся алгоритм действий:

1. Marshmallow не умеет работать с объектом типа bson.ObjectId, поэтому ее нужно научить этому. Для этого описываем новое поле ObjectIdField, которое наследуется от базового класса fields.Field, и реализуем 2 метода: *_deserialize* и *_serialize.* _deserialize вызывается, когда данные нужно превратить из словаря в объект dataclass, а _serialize — когда из объекта dataclass нужно получить словарь
2. Чтобы в описании dataclass использовать тип bson.ObjectId, нужно явно указать, какое поле может обрабатывать такой тип данных. Для этого мы добавляем в TYPE_MAPPING описанный ранее класс ObjectIdField: Schema.TYPE_MAPPING[bson.ObjectId] = ObjectIdField
3. Описываем объект User так же, как делали в главе про aiohttp

Как пользоваться описанным dataclass

Пример:

```
import asyncio
import os
from dataclasses import asdict
from typing import Optional

from bson import ObjectId
from motor.motor_asyncio import AsyncIOMotorClient
from .schema import User


def get_client():
    return AsyncIOMotorClient(os.getenv("MONGO_URL"))


def get_db():
    return get_client().metaclass


def get_user_collection():
    return get_db().user


async def insert_user(user: User) -> ObjectId:
    document = asdict(user)
    del document['_id']
    result = await get_user_collection().insert_one(document)
    return result.inserted_id


async def find_user_by_id(user_id: ObjectId) -> Optional[User]:
    document = await get_user_collection().find_one({'_id': user_id})
    if not document:
        return None
    return User.Schema().load(document)


async def run():
    user_id = await insert_user(
        User(first_name='Alexander', last_name='Opryshko', is_tutor=True)
    )
    print('insert_user', user_id)
    user = await find_user_by_id(user_id)
    print('find_user_by_id', user)


if __name__ == "__main__":
    asyncio.run(run())
```

Пример похож на предыдущий, за исключением нескольких нюансов:

- *insert_user* принимает объект пользователя. Из него нужно получить словарь. Для этого можно воспользоваться функцией asdict, которая встроена в библиотеку dataclass, или использовать dump из библиотеки marshmallow.
- В примере используется способ через asdict. Далее из полученного словаря нужно удалить _id, чтобы Mongo автоматически сгенерировала _id для документа.
- find_user_by_id: функция collection.find_one() вернет словарь. Его нужно превратить в объект User, поэтому нужно вызвать User.Schema().load(document)